Programing with Multiple Processes in C

fork, wait, execlp, WIFEXITED, WEXITSTATUS, file operations, and make

Assignment Information

- Four executables will be needed
  - **Generator** – Main program, that opens, reads the characters and closes the file, forks child processes.
  - ***Generator.c, OddEven.c, PerfectSquare.c and Factorial.c.***

Outline

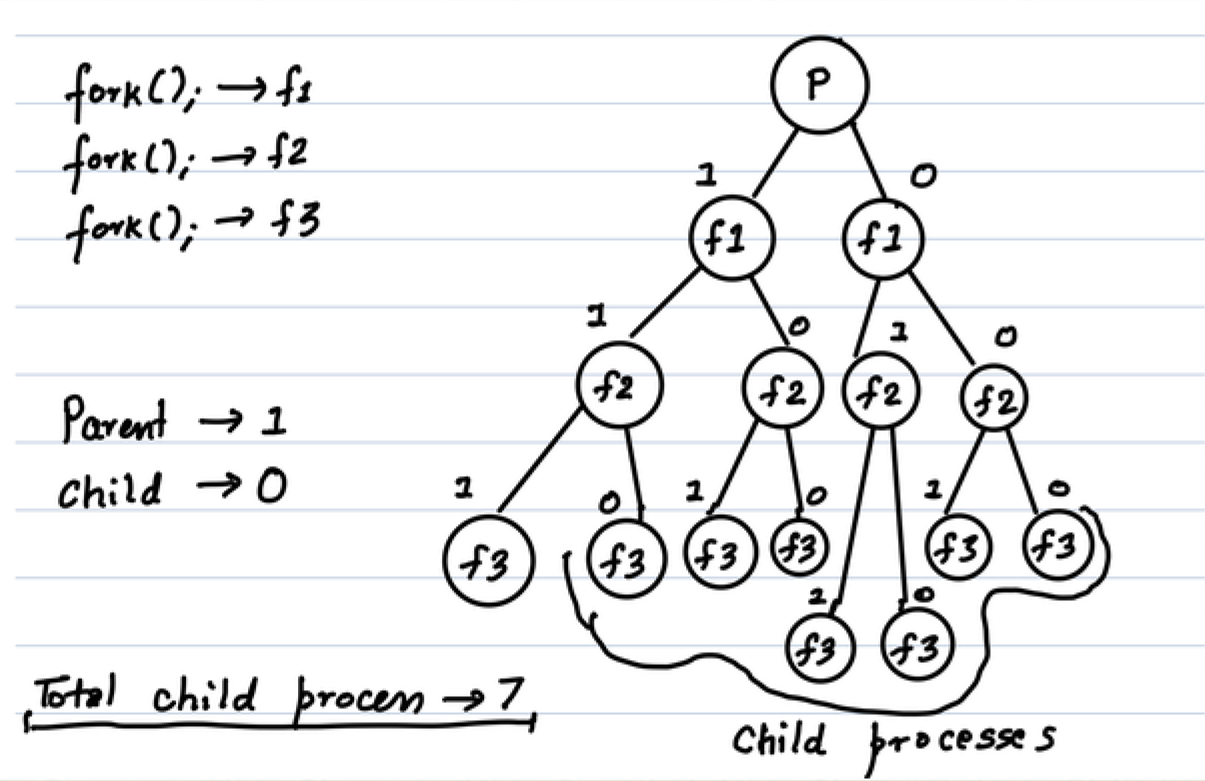
- Learn how to use the following
  - Passing command line argument to Main Program
  - File Operation (fopen, perror, fgets, sizeof, strcspn, atoi)
  - Creating new child process (fork, perror)
  - Executing the program in the child process, passing argument as a command-line argument (execlp)
  - Waiting for the child process to terminate and (wait)
  - Checking if the child process terminated normally (WIFEXITED)
  - Extracting the exit status of the child process (WEXITSTATUS)

Flowchart

fork()

- Generates an exact copy of parent process except for the value it returns.
- Both Processes continue to work after the fork() execution.
- In a child process, fork() returns zero
- In the parent process it will return the child's process ID
- If return value is -1, then fork() failed.
- Any process can retrieve its process ID with getpid().
- Syntax:
  - pid_t pid=fork();

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
fork();
fork();
fork();
printf("hello\n");
return 0;
}
```



wait()

- Makes parent process wait until the child has been entirely executed .
- Use WIFEXITED() to check whether child process has terminated normally, as opposed to dying with a signal .
- Use WEXITSTATUS() to retrieve return value of child process
- Syntax: pid_t wait(int *stat_loc);

execlp()

- Executes a new program within a child process
- Arguments passed - the name of the executable and filename like "./Starter", ”Starter"
- Also pass any needed command line arguments as parameters
- Terminate list of arguments with NULL
- Syntax
    - *int execlp(const char * file, const char * arg0, const char * arg1, … const char * argn, NULL);*
    - *execlp(“./Fibonacci”, “Fibonacci”, arg_str, NULL);*

File Operations

- We need several functions for this assignment.
- They are:
  - fopen()
  - fclose()
  - fgets() or fgetc()

fopen()

- Used to open a file, whose name is given as the argument.
- It returns a pointer to the opened file.
- Syntax:
  - FILE * fp = fopen(const char *filename, const char *mode)

fclose()

- Closes the stream to the file.
- Buffers are flushed.
- Syntax
  - int fclose(FILE *stream)

fgets()

- Reads a line from a file
- Puts the line into the provided array/string
- Syntax:

```
int fgets(char *s, int size, FILE *stream)
```

- Use:

```
char buf[256];
while (fgets(buf, sizeof(buf), in)
        // deal with the string in buf
```

Why use make?

- Enables developers to easily compile large and complex programs with many components.
- Situation: There are thousands of lines of code, distributed in multiple source files, written by many developers and arranged in several sub-directories. This project also contains several component divisions and these components may have complex inter-dependencies.

Variable assignments in make

- By convention, predefined variable names used in a Makefile are in upper case, and user-defined variables are lower case.

Example: CC = gcc

- We can use the value assigned later as $()

Example: $(CC)

Makefile Structure

- Makefile contains definitions and rules.
- A definition has the form:

  VAR = value

- A rule has the form:

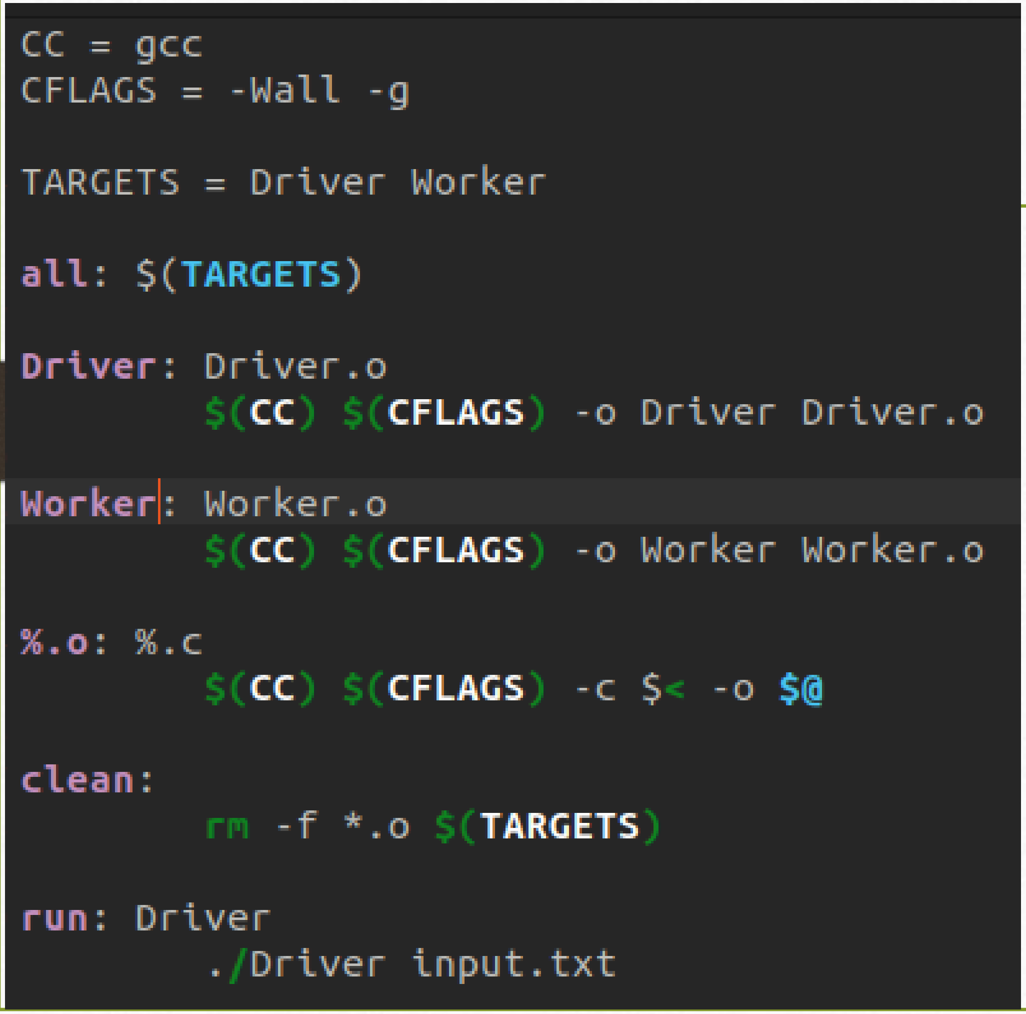
```
Output files: input files
<tab>Commands to turn inputs to outputs
```

- All commands must be tab-indented. Spaces don't work!
- The make <target> command executes the rule with the <target>. If target not is specified, it defaults to the first rule defined in the Makefile.

Patterns and Special variables

- % : Wildcard pattern-matching, for generic targets.
- $@ : Full target name of the current target.
- $? : Returns the dependencies that are newer than the current target.
- $* : Returns the text that corresponds to % in the target.
- $< : Name of the first dependency.
- $^ : Name of the all dependencies with space as the delimiter.

Demo Makefiles



```
CC = gcc
CFLAGS = -Wall -g

TARGETS = Driver Worker

all: $(TARGETS)

Driver: Driver.o
        $(CC) $(CFLAGS) -o Driver Driver.o

Worker: Worker.o
        $(CC) $(CFLAGS) -o Worker Worker.o

%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@

clean:
        rm -f *.o $(TARGETS)

run: Driver
        ./Driver input.txt
```

CC = gcc

- This line defines a **variable** called `CC that assigns gcc which represents C compiler`

`CFLAGS = -Wall -g`

- `This defines the compiler flags that will be passed to gcc during compilation.`
  - `-Wall: This enables all common compiler warnings`
  - `-g`: This flag includes **debugging information** in the compiled binaries

TARGETS = Driver Worker

- This defines a variable `TARGETS`, which contains the **list of final executables** that the `Makefile` will produce: `Driver`, `Worker`.

all: $(TARGETS)

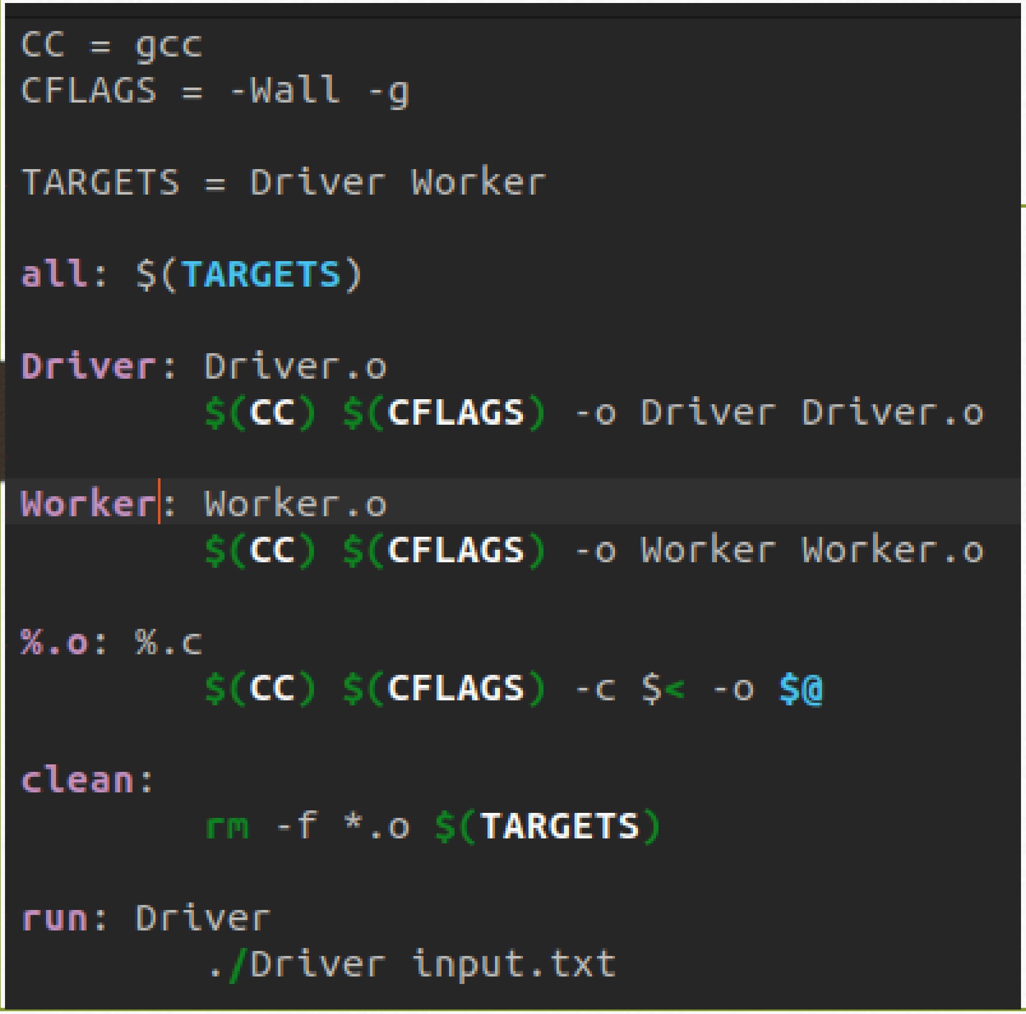
- This defines the **default target** (named `all`), which will be executed if no specific target is given when running `make`.

Driver: Driver.o

$(CC) $(CFLAGS) -o Driver Driver.o

- This rule defines how to build the `Driver` executable.
  - This is the command to **link** the `Driver.o` object file and produce the final `Driver` executable.
  - `$(CC)` is `gcc`, and `$(CFLAGS)` are the compiler flags (`-Wall -g`).
  - `-o Driver` specifies the output file, which will be named `Driver`.

Demo Makefiles



```
CC = gcc
CFLAGS = -Wall -g

TARGETS = Driver Worker

all: $(TARGETS)

Driver: Driver.o
	$(CC) $(CFLAGS) -o Driver Driver.o

Worker: Worker.o
	$(CC) $(CFLAGS) -o Worker Worker.o

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f *.o $(TARGETS)

run: Driver
	./Driver input.txt
```

%.o: %.c

- This is a **pattern rule** that defines how to compile any `.c` file into a `.o` (object) file.
- `%.o` and `%.c` are placeholders (wildcards), where `make` will substitute the `%` with the actual file name (e.g., `Driver.c` to `Driver.o`).
- This rule applies to all the `.c` files without having to explicitly list each file.

$(CC) $(CFLAGS) -c $< -o $@

- This is the command to compile a `.c` file into an object file.
- `$(CC)` is `gcc`, and `$(CFLAGS)` are the compiler flags.
- `-c` tells the compiler to **compile only** (i.e., generate an object file, not a full executable).
- `$<` represents the **first prerequisite** (in this case, the `.c` file being compiled).
- `$@` represents the **target** (in this case, the `.o` file being generated).

clean:

rm -f *.o $(TARGETS)

- This command removes all `.o` files and the target executables

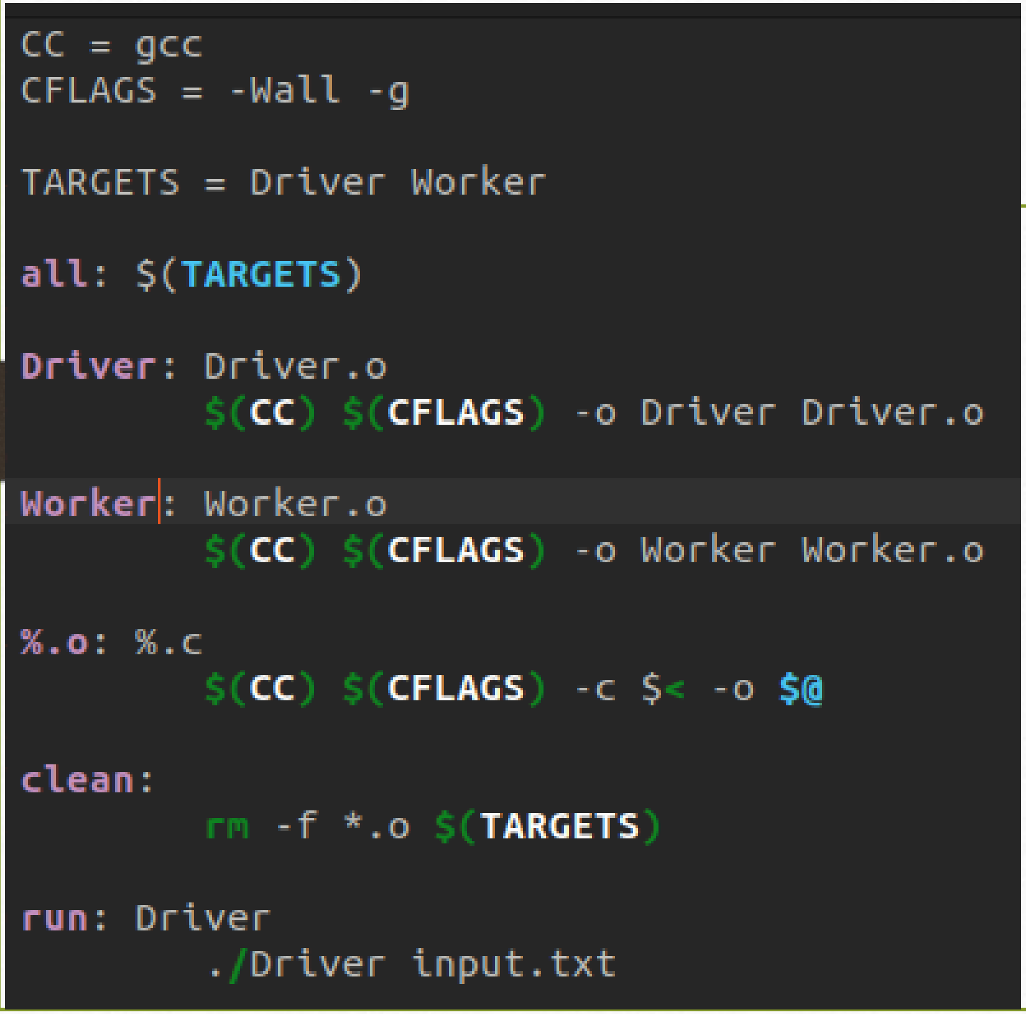
run: Driver

- This rule defines a `run` target, which depends on the `Driver` executable.
- It will ensure that `Driver` is built before attempting to run it.

./Driver input.txt

- This is the command to **run the `Driver` program** with `input.txt` as the command-line argument.

Demo Makefiles



```
CC = gcc
CFLAGS = -Wall -g

TARGETS = Driver Worker

all: $(TARGETS)

Driver: Driver.o
        $(CC) $(CFLAGS) -o Driver Driver.o

Worker: Worker.o
        $(CC) $(CFLAGS) -o Worker Worker.o

%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@

clean:
        rm -f *.o $(TARGETS)

run: Driver
        ./Driver input.txt
```

When you run `make`:

- `make` starts with the `all` target and builds the required executables (`Generator`, `Fibonacci`, `Perrin`, `Composite`).
- It checks the `.c` files for changes, compiles them into `.o` files, and then links them into executables.

To clean up the project:

- You can run `make clean`, which removes all the generated object files and executables.

To run the `Generator` program:

- You can use `make run`, which will build `Generator` if necessary, and then execute it with the argument `input.txt`.

Thank You

Acknowledgements

- These slides are based on contributions of current and past CS370 instructors and TAs, including J. Applin,A. Yeluri, Y. K. Malaiya, Phil sharp and S. Pallickara.